

Lattice based Least Fixed Point Logic

Piotr Filipiuk, Flemming Nielson, and Hanne Riis Nielson

DTU Informatics, Richard Petersens Plads, Technical University of Denmark,
DK-2800 Kongens Lyngby, Denmark
{pifi,nielson,riis}@imm.dtu.dk

Abstract. As software systems become more complex, there is an increasing need for new static analyses. Thanks to the declarative style, logic programming is an attractive formalism for specifying them. However, prior work on using logic programming for static analysis focused on analyses defined over some powerset domain, which is quite limiting. In this paper we present a logic that lifts this restriction, called Lattice based Least Fixed Point Logic (LLFP), that allows interpretations over any complete lattice satisfying Ascending Chain Condition. The main theoretical contribution is a Moore Family result that guarantees that there always is a unique least solution for a given problem. Another contribution is the development of solving algorithm that computes the least model of LLFP formulae guaranteed by the Moore Family result.

Keywords: Static analysis, logic programming, abstract interpretation.

1 Introduction

Nowadays, we heavily rely on software systems. At the same time they become bigger and more complex, and hence the number of potential errors increases. In order to achieve more reliable systems, formal verification techniques may be applied. A widely used verification technique is static analysis, which reasons about system behavior without executing it. It is performed statically at compile-time, and it computes safe approximations of values or behaviors that may occur at run-time. Static analysis is increasingly recognized as a fundamental technique for program verification, bug detection, compiler optimization and software understanding.

Unfortunately, developing new static analyses is difficult and error-prone. In order to overcome that problem it is desirable to implement prototypes of analyses that are easy to analyse for complexity and correctness. Since analysis specifications are generally written in a declarative style, logic programming presents an attractive model for producing executable specifications of analyses. Furthermore, thanks to the advances in logic programming, the associated solvers became more efficient.

In this paper we present a framework that facilitates rapid prototyping of new static analyses. The approach taken falls within the Abstract Interpretation [8,7] framework, thus there always is a unique best solution to the analysis problem

considered. The framework consists of the Lattice based Least Fixed Point Logic (LLFP) and the associated solver. The most prominent feature of the LLFP logic is its interpretation over complete lattices satisfying the Ascending Chain Condition, which makes it possible to express sophisticated analyses in LLFP. The solver combines a continuation passing style algorithm with propagation of differences, and uses prefix trees as its main data structure. The applicability of the framework is illustrated by presenting a specification of interval analysis, which could not be specified using logics traditionally used such as Datalog [1,5] or ALFP [18].

This paper is organized as follows. In Section 2 we present the problem we want to solve and indicate a solution. Section 3 introduces syntax and semantics of the LLFP logic. In Section 4 we establish our main theoretical contribution; namely a Moore Family result for LLFP. Section 5 describes the solving algorithm for LLFP. We conclude and discuss future work in Section 6.

2 The problem

There is an immense body of work on using logic for specifying static analyses. However, logics traditionally used have some limitations. To illustrate the problem, let us briefly introduce the Alternation free Least Fixed Point Logic (ALFP) and then try to devise the ALFP specifications of two analyses: detection of signs and interval analysis.

Alternation-free Least Fixed Point Logic. Many static analyses can be succinctly expressed using Alternation-free Least Fixed Point Logic (ALFP) [18]. The logic is a generalization of Datalog [1,5] and it has proved to have a number of properties essential for specifying static analyses such as the existence of a unique least model. The syntax of ALFP is given by

$$\begin{aligned} v &::= x \mid a \\ pre &::= R(v_1, \dots, v_k) \mid \neg R(v_1, \dots, v_k) \mid pre_1 \wedge pre_2 \\ &\quad \mid pre_1 \vee pre_2 \mid \exists x : pre \mid v_1 = v_2 \mid v_1 \neq v_2 \\ cl &::= R(v_1, \dots, v_k) \mid \mathbf{1} \mid cl_1 \wedge cl_2 \mid pre \Rightarrow cl \mid \forall x : cl \end{aligned}$$

where we write a for constants, x for analysis variables, v for values, R for predicates, pre for preconditions, and cl for clauses. The clauses are interpreted over a universe \mathcal{U} of constants, $a \in \mathcal{U}$. The interpretation is given in terms of satisfaction relations $(\rho, \sigma) \models pre$ and $(\rho, \sigma) \models cl$ where ρ is an interpretation of predicates, and σ is an interpretation of variables. The definition is standard and hence omitted. Due to the use of negation, we impose a *stratification* condition similar to the one in Datalog [1,5]. This intuitively means that no predicate depends on the negation of itself. We refer to [18] for more details.

Example 1. Using the notion of stratification we can define equality E and non-equality N predicates as follows

$$(\forall x : E(x, x)) \wedge (\forall x : \forall y : \neg E(x, y) \Rightarrow N(x, y))$$

The formula is stratified, since predicate E is fully asserted before it is negatively queried in the clause asserting predicate N .

Detection of signs analysis. Now, let us consider the ALFP formulation of the detection of signs analysis. The analysis aims to determine for each program point and each variable, the possible sign (negative, zero or positive) that the variable may have whenever the execution reaches that point. In the following, we use program graphs as representation of the program under consideration [2]. Compared to the classical flow graphs [14,16], the main difference is that in the program graphs the actions label the edges rather than the states. Here we focus on three types of actions: assignments, boolean expressions and the skip action. For simplicity we assume that assignments are in three-address form. The analysis is defined by predicate A , and we begin with initializing the initial state of the program graph, q_0 , with all possible signs for all variables v occurring in the underlying program graph

$$\bigwedge_{v \in Var} A(q_0, v, -) \wedge A(q_0, v, 0) \wedge A(q_0, v, +)$$

Intuitively it indicates that at state q_0 all variables may have all possible values. Now we consider the ALFP specifications for each type of action. Whenever we have $q_s \xrightarrow{x:=y \star z} q_t$ in the program graph we generate

$$\begin{aligned} \forall s : \forall s_y : \forall s_z : A(q_s, y, s_y) \wedge A(q_s, z, s_z) \wedge R_\star(s_y, s_z, s) \Rightarrow A(q_t, x, s) \wedge \\ \forall v : \forall s : v \neq x \wedge A(q_s, v, s) \Rightarrow A(q_t, v, s) \end{aligned}$$

where we assume that we have a relation for each type of arithmetic operation, denoted by R_\star in the above formula. The first conjunct states that for all possible values s , s_y and s_z , if at state q_s the signs of variables y and z are s_y and s_z , respectively, and the sign of the result of evaluating the arithmetic operation \star is s , then at state q_t variable x will have sign s . The second conjunct expresses that for all variables v and signs s , if the variable is different than x and at state q_s it has sign s , then it will have the same sign at state q_t . Similarly, whenever we have $q_s \xrightarrow{e} q_t$ or $q_s \xrightarrow{skip} q_t$ in the program graph, we generate a clause

$$\forall v : \forall s : A(q_s, v, s) \Rightarrow A(q_t, v, s)$$

The clause simply propagates the signs of all variables along the edge of the program graph, without altering it.

In a similar manner we could formulate other analyses such as pointer analysis [21,15,20,4], or classical data flow analyses [13,19]. In more general terms, logics traditionally used, e.g. Datalog and ALFP, can be used for specifying analyses defined over a powerset domain. However, as we show in the next paragraph, many interesting analyses are defined over some mathematical structure such as a complete lattice. Thus, let us now consider interval analysis as an example of such an analysis.

Interval analysis. The purpose of interval analysis is to determine for each program point an interval containing possible values of variables whenever that point is reached during run-time execution. The analysis results can be used for Array Bound Analysis, which determines whether an array index is always within the bounds of the array. If this is the case, a run-time check can safely be eliminated, which makes code more efficient.

We begin with defining the complete lattice $(Interval, \sqsubseteq_I)$ over which the analysis is defined. The underlying set is

$$Interval = \perp \cup \{[z_1, z_2] \mid z_1 \leq z_2, z_1 \in Z \cup \{-\infty\}, z_2 \in Z \cup \{\infty\}\}$$

where Z is a finite subset of integers, $Z \subseteq \mathbb{Z}$, and the integer ordering \leq on \mathbb{Z} is extended to an ordering on $Z' = Z \cup \{-\infty, \infty\}$ by taking for all $z \in Z$: $-\infty \leq z$, $z \leq \infty$ and $-\infty \leq \infty$. In the above definition, \perp denotes an empty interval, whereas $[z_1, z_2]$ is the interval from z_1 to z_2 including the end points, where $z_1, z_2 \in Z$. The interval $[-\infty, \infty]$ is equivalent to the top element, \top . In the following we use i to denote an interval from $Interval$. The partial ordering \sqsubseteq_I in $Interval$ uses operations \inf and \sup

$$\inf(i) = \begin{cases} \infty & \text{if } i = \perp \\ z_1 & \text{if } i = [z_1, z_2] \end{cases} \quad \sup(i) = \begin{cases} -\infty & \text{if } i = \perp \\ z_2 & \text{if } i = [z_1, z_2] \end{cases}$$

and is defined as

$$i_1 \sqsubseteq_I i_2 \text{ iff } \inf(i_2) \leq \inf(i_1) \wedge \sup(i_1) \leq \sup(i_2)$$

The intuition behind the partial ordering \sqsubseteq_I in $Interval$ is that

$$i_1 \sqsubseteq_I i_2 \Leftrightarrow \{z \mid z \text{ belongs to } i_1\} \subseteq \{z \mid z \text{ belongs to } i_2\}$$

Unfortunately, due to limited expressiveness of the ALFP logic (and similarly Datalog), interval analysis can not be specified using these formalisms. Hence, in the following section we present a solution for that problem; namely we introduce LLFP logic.

3 Syntax and Semantics

In the previous section we briefly introduced ALFP logic, which is interpreted over a finite universe of atoms; in this section we present an extension of ALFP called LLFP allowing interpretations over complete lattices satisfying Ascending Chain Condition. We also allow function terms as arguments of relations. Since functions over the universe \mathcal{U} can be represented as relations, we do not consider them here. Instead, we focus on functions over a complete lattice $\llbracket f \rrbracket : \mathcal{L}^k \rightarrow \mathcal{L}$, and we restrict our attention to monotone functions only. Recall that a function $\llbracket f \rrbracket : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ between partially ordered sets $\mathcal{L}_1 = (\mathcal{L}_1, \sqsubseteq_1)$ and $\mathcal{L}_2 = (\mathcal{L}_2, \sqsubseteq_2)$ is monotone if

$$\forall l, l' \in \mathcal{L}_1 : l \sqsubseteq_1 l' \Rightarrow \llbracket f \rrbracket(l) \sqsubseteq_2 \llbracket f \rrbracket(l')$$

Let us begin with introducing necessary definitions.

Definition 1. A complete lattice $\mathcal{L} = (\mathcal{L}, \sqsubseteq) = (\mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ is a partially ordered set $(\mathcal{L}, \sqsubseteq)$ such that all subsets have least upper bounds as well as greatest lower bounds.

A subset $Y \subseteq \mathcal{L}$ of a partially ordered set $\mathcal{L} = (\mathcal{L}, \sqsubseteq)$ is a chain if

$$\forall l_1, l_2 \in Y : (l_1 \sqsubseteq l_2) \vee (l_2 \sqsubseteq l_1)$$

Hence, a chain is a (possibly empty) subset of \mathcal{L} that is totally ordered. A sequence $(l_n)_n$ of elements in \mathcal{L} is an ascending chain if

$$n < m \Rightarrow l_n \sqsubseteq l_m$$

We say that a sequence $(l_n)_n$ eventually stabilises if and only if

$$\exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : n > n_0 \Rightarrow l_n = l_{n_0}$$

The partially ordered set \mathcal{L} satisfies Ascending Chain Condition if and only if all ascending chains eventually stabilise. Essentially, the Ascending Chain Condition guarantees that the least fixed point computation always terminates. Due to the use of negation in the logic, we need to introduce a complement operator, \mathbb{C} , in the underlying complete lattice. The only condition that we impose on the complement is anti-monotonicity i.e. $\forall l_1, l_2 \in \mathcal{L} : l_1 \sqsubseteq l_2 \Rightarrow \mathbb{C}l_1 \sqsupseteq \mathbb{C}l_2$, which is necessary for establishing Moore Family result. The following definition introduces the syntax of LLFP.

Definition 2. Given fixed countable and pairwise disjoint sets \mathcal{X} and \mathcal{Y} of variables, a non-empty and finite universe \mathcal{U} , a complete lattice satisfying Ascending Chain Condition \mathcal{L} , finite alphabets \mathcal{R} and \mathcal{F} of predicate and function symbols, respectively, we define the set of LLFP formulae (or clause sequences), *cls*, together with clauses, *cl*, preconditions, *pre*, terms *u* and lattice terms *V* and *V'* by the grammar:

$$\begin{aligned} u &::= x \mid a \\ V &::= Y \mid [u] \\ V' &::= V \mid f(V') \\ pre &::= R(\mathbf{u}; V) \mid \neg R(\mathbf{u}; V) \mid Y(u) \mid pre_1 \wedge pre_2 \mid pre_1 \vee pre_2 \\ &\quad \mid \exists x : pre \mid \exists Y : pre \\ cl &::= R(\mathbf{u}; V') \mid \mathbf{1} \mid cl_1 \wedge cl_2 \mid pre \Rightarrow cl \mid \forall x : cl \mid \forall Y : cl \\ cls &::= cl_1, \dots, cl_s \end{aligned}$$

Here $x \in \mathcal{X}$, $a \in \mathcal{U}$, $Y \in \mathcal{Y}$, $R \in \mathcal{R}$, $f \in \mathcal{F}$, and $s \geq 1$. Furthermore, \mathbf{u} and \mathbf{V}' abbreviate tuples (u_1, \dots, u_k) and (V'_1, \dots, V'_k) for some $k \geq 0$, respectively.

We write $fv(\cdot)$ for the set of free variables in the argument \cdot . Occurrences of $R(\mathbf{u}; V)$ and $\neg R(\mathbf{u}; V)$ in preconditions are called positive, resp. negative, queries and we require that $fv(\mathbf{u}) \subseteq \mathcal{X}$ and $fv(V) \subseteq \mathcal{Y} \cup \mathcal{X}$; these variables are *defining* occurrences. Occurrences of $Y(u)$ in preconditions must satisfy $Y \in \mathcal{Y}$ and $fv(u) \subseteq \mathcal{X}$; Y is an *applied* occurrence, u is a defining occurrence. Clauses

of the form $R(\mathbf{u}; V')$ are called *assertions*; we require that $fv(\mathbf{u}) \subseteq \mathcal{X}$ and $fv(V') \subseteq \mathcal{Y} \cup \mathcal{X}$ and we note that these variables are applied occurrences. A clause cl satisfying these conditions together with $fv(cl) = \emptyset$ is said to be *well-formed*; we are only interested in clause sequences cls consisting of well-formed clauses.

In order to ensure desirable theoretical and pragmatic properties in the presence of negation, we impose a notion of *stratification* similar to the one in Datalog [1,5]. Intuitively, stratification ensures that a negative query is not performed until the predicate has been fully asserted. This is important for ensuring that once a precondition evaluates to true it will continue to be true even after further assertions of predicates.

Definition 3. *The formula $cls = cl_1, \dots, cl_s$ is stratified if there exists a function $\text{rank} : \mathcal{R} \rightarrow \{0, \dots, s\}$ such that for all $i = 1, \dots, s$:*

- $\text{rank}(R) = i$ for every assertion R in cl_i ;
- $\text{rank}(R) \leq i$ for every positive query R in cl_i ; and
- $\text{rank}(R) < i$ for every negative query $\neg R$ in cl_i .

The following example illustrates the use of negation in the LLFP formula.

Example 2. Similarly to Example 1, we can define equality E and non-equality N predicates in LLFP as follows

$$(\forall x : E(x; [x])), (\forall x : \forall Y : \neg E(x; Y) \Rightarrow N(x; Y))$$

According to Definition 3 the formula is stratified, since predicate E is fully asserted before it is negatively queried in the clause asserting predicate N . As a result we can dispense with an explicit treatment of $=$ and \neq in the development that follows. On the other hand the Definition 3 rules out

$$(\forall x : \forall Y : \neg P(x; Y) \Rightarrow Q(x; Y)), (\forall x : \forall Y : \neg Q(x; Y) \Rightarrow P(x; Y))$$

To specify the semantics of LLFP we introduce the interpretations ϱ , ς and ζ of predicate symbols, variables and function symbols, respectively. Formally we have

$$\begin{aligned} \varrho &: \prod_k \mathcal{R}_{/k} \rightarrow \mathcal{U}^k \rightarrow \mathcal{L} \\ \varsigma &: (\mathcal{X} \rightarrow \mathcal{U}) \times (\mathcal{Y} \rightarrow \mathcal{L}_{\neq \perp}) \\ \zeta &: \prod_k \mathcal{F}_{/k} \rightarrow \mathcal{L}^k \rightarrow \mathcal{L} \end{aligned}$$

In the above $\mathcal{R}_{/k}$ stands for a set of predicate symbols of arity k , and \mathcal{R} is a disjoint union of $\mathcal{R}_{/k}$, hence $\mathcal{R} = \bigsqcup_k \mathcal{R}_{/k}$. Similarly, $\mathcal{F}_{/k}$ is a set of function symbols of arity k over the complete lattice \mathcal{L} . The set \mathcal{F} is then defined as disjoint unions of $\mathcal{F}_{/k}$; hence $\mathcal{F} = \bigsqcup_k \mathcal{F}_{/k}$. The interpretation of variables from \mathcal{X} is given by $\llbracket x \rrbracket(\zeta, \varsigma) = \varsigma(x)$, where $\varsigma(x)$ is the element from \mathcal{U} bound to $x \in \mathcal{X}$. Analogously, the interpretation of variables from \mathcal{Y} is given by $\llbracket Y \rrbracket(\zeta, \varsigma) = \varsigma(Y)$, where $\varsigma(Y)$ is the element from $\mathcal{L}_{\neq \perp} = \mathcal{L} \setminus \{\perp\}$ bound to $Y \in \mathcal{Y}$. We do not allow variables from \mathcal{Y} to be mapped to \perp in order to establish a relationship

between ALFP and LLFP in the case of powerset lattice, i.e. $\mathcal{P}(\mathcal{U})$, which we briefly describe later. In order to give the interpretation of $[u]$, we introduce a function $\beta : \mathcal{U} \rightarrow \mathcal{L}$. The β function is called a *representation function* and the idea is that β maps a value from the universe \mathcal{U} to the *best* property describing it. For example in the case of a powerset lattice, β could be defined by $\beta(a) = \{a\}$ for all $a \in \mathcal{U}$. Then the interpretation is given by $\llbracket [u] \rrbracket(\zeta, \varsigma) = \beta(\llbracket u \rrbracket(\zeta, \varsigma))$. The interpretation of function terms is defined as $\llbracket f(\mathbf{V}') \rrbracket(\zeta, \varsigma) = \zeta(f)(\llbracket \mathbf{V}' \rrbracket(\zeta, \varsigma))$. For the functions we require that $\zeta(f) : \mathcal{L}^k \rightarrow \mathcal{L}$ is monotone. The interpretation of terms is generalized to sequences \mathbf{u} of terms in a point-wise manner by taking $\llbracket a \rrbracket(\zeta, \varsigma) = a$ for all $a \in \mathcal{U}$, thus $\llbracket (u_1, \dots, u_k) \rrbracket(\zeta, \varsigma) = (\llbracket u_1 \rrbracket(\zeta, \varsigma), \dots, \llbracket u_k \rrbracket(\zeta, \varsigma))$. The interpretation of lattice terms V (and V') is generalized to sequences \mathbf{V} (and \mathbf{V}') of lattice terms in the similar way.

The satisfaction relations for preconditions *pre*, clauses *cl* and clause sequences *cls* are specified by:

$$(\varrho, \varsigma) \models_{\beta} pre, \quad (\varrho, \zeta, \varsigma) \models_{\beta} cl \quad \text{and} \quad (\varrho, \zeta, \varsigma) \models_{\beta} cls$$

The formal definition is given in Table 1; here $\varsigma[x \mapsto a]$ stands for the mapping that is as ς except that x is mapped to a and similarly $\varsigma[Y \mapsto l]$ stands for the mapping that is as ς except that Y is mapped to $l \in \mathcal{L}_{\neq \perp}$.

Table 1. Semantics of LLFP

$(\varrho, \varsigma) \models_{\beta} R(\mathbf{u}; V)$	<u>iff</u> $\varrho(R)(\varsigma(\mathbf{u})) \supseteq \varsigma(V)$
$(\varrho, \varsigma) \models_{\beta} \neg R(\mathbf{u}; V)$	<u>iff</u> $\complement(\varrho(R)(\varsigma(\mathbf{u}))) \supseteq \varsigma(V)$
$(\varrho, \varsigma) \models_{\beta} Y(u)$	<u>iff</u> $\beta(\varsigma(u)) \sqsubseteq \varsigma(Y)$
$(\varrho, \varsigma) \models_{\beta} pre_1 \wedge pre_2$	<u>iff</u> $(\varrho, \varsigma) \models_{\beta} pre_1$ and $(\varrho, \varsigma) \models_{\beta} pre_2$
$(\varrho, \varsigma) \models_{\beta} pre_1 \vee pre_2$	<u>iff</u> $(\varrho, \varsigma) \models_{\beta} pre_1$ or $(\varrho, \varsigma) \models_{\beta} pre_2$
$(\varrho, \varsigma) \models_{\beta} \exists x : pre$	<u>iff</u> $(\varrho, \varsigma[x \mapsto a]) \models_{\beta} pre$ for some $a \in \mathcal{U}$
$(\varrho, \varsigma) \models_{\beta} \exists Y : pre$	<u>iff</u> $(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} pre$ for some $l \in \mathcal{L}_{\neq \perp}$
$(\varrho, \zeta, \varsigma) \models_{\beta} R(\mathbf{u}; V')$	<u>iff</u> $\varrho(R)(\llbracket \mathbf{u} \rrbracket(\zeta, \varsigma)) \supseteq \llbracket V' \rrbracket(\zeta, \varsigma)$
$(\varrho, \zeta, \varsigma) \models_{\beta} \mathbf{1}$	<u>iff</u> true
$(\varrho, \zeta, \varsigma) \models_{\beta} cl_1 \wedge cl_2$	<u>iff</u> $(\varrho, \zeta, \varsigma) \models_{\beta} cl_1$ and $(\varrho, \zeta, \varsigma) \models_{\beta} cl_2$
$(\varrho, \zeta, \varsigma) \models_{\beta} pre \Rightarrow cl$	<u>iff</u> $(\varrho, \zeta, \varsigma) \models_{\beta} cl$ whenever $(\varrho, \varsigma) \models_{\beta} pre$
$(\varrho, \zeta, \varsigma) \models_{\beta} \forall x : cl$	<u>iff</u> $(\varrho, \zeta, \varsigma[x \mapsto a]) \models_{\beta} cl$ for all $a \in \mathcal{U}$
$(\varrho, \zeta, \varsigma) \models_{\beta} \forall Y : cl$	<u>iff</u> $(\varrho, \zeta, \varsigma[Y \mapsto l]) \models_{\beta} cl$ for all $l \in \mathcal{L}_{\neq \perp}$
$(\varrho, \zeta, \varsigma) \models_{\beta} cl_1, \dots, cl_s$	<u>iff</u> $(\varrho, \zeta, \varsigma) \models_{\beta} cl_i$ for all $i, 1 \leq i \leq s$

Relationship to ALFP. As reader may have already noticed, in the case the underlying complete lattice is $\mathcal{P}(\mathcal{U})$ the two logics are essentially equivalent. More precisely, in the case of powerset lattice, $\mathcal{P}(\mathcal{U})$, function β given by $\beta(a) = \{a\}$ for all $a \in \mathcal{U}$, and without function terms we can translate LLFP formula

into a corresponding ALFP one and vice versa. Intuitively, we get the following correspondence between interpretations of relations

$$\forall \mathbf{a}, b : ((\mathbf{a}, b) \in \rho(R) \Leftrightarrow \varrho(R)(\mathbf{a}) \supseteq \{b\})$$

The idea is that a relation R in LLFP with interpretation $\varrho(R) \in \mathcal{U}^k \rightarrow \mathcal{P}(\mathcal{U})$ is replaced by a relation in ALFP (also named R) with interpretation $\rho(R) \in \mathcal{P}(\mathcal{U}^{k+1})$. Note that if $\varrho(R)(\mathbf{a}) = \perp$ then $\rho(R)$ does not contain any tuples with \mathbf{a} as the first k components.

Interval analysis in LLFP. Now let us give an LLFP specification of interval analysis. The analysis is defined by the predicate A . Similarly to Datalog or ALFP, the specification is defined over a universe \mathcal{U} , which in this case is a set of all variables, Var , appearing in the program as well as states in the underlying program graph. In addition, the LLFP logic allows interpretations over complete lattices satisfying Ascending Chain Condition. Here we use the lattice $(Interval, \sqsubseteq_I)$, defined in Section 2.

The specification consists of the initialization clauses and clauses corresponding to three types of actions in the underlying program graph. First, for the initial state, q_0 , we initialize all variables in the program graph with the \top element, denoting that they may have all possible values

$$\bigwedge_{v \in Var} A(q_0, v; \top)$$

Furthermore, whenever we have $q_s \xrightarrow{x:=y \star z} q_t$ in the program graph we generate

$$\begin{aligned} \forall i_y : \forall i_z : A(q_s, y; i_y) \wedge A(q_s, z; i_z) \Rightarrow A(q_t, x; f_\star(i_y, i_z)) \wedge \\ \forall v : \forall i : v \neq x \wedge A(q_s, v; i) \Rightarrow A(q_t, v; i) \end{aligned}$$

The first conjunct updates the possible interval of values for the assigned variable (in that case for variable x), with the result of evaluating the arithmetic operation $y \star z$. The second conjunct propagates the analysis information for all variables except variable x without altering it. Furthermore, whenever we have $q_s \xrightarrow{e} q_t$ or $q_s \xrightarrow{skip} q_t$ in the program graph, we generate a clause

$$\forall v : \forall i : A(q_s, v, i) \Rightarrow A(q_t, v, i)$$

which simply propagates the analysis information along the edge of the program graph, without making any changes.

4 Moore family result for LLFP

In this section we establish a Moore family result for LLFP that guarantees that there always is a unique best solution for LLFP clauses.

Definition 4. A Moore family is a subset Y of a complete lattice $\mathcal{L} = (\mathcal{L}, \sqsubseteq)$ that is closed under greatest lower bounds: $\forall Y' \subseteq Y : \bigcap Y' \in Y$.

It follows that a Moore family always contains a least element, $\bigcap Y$, and a greatest element, $\bigcap \emptyset$, which equals the greatest element, \top , from \mathcal{L} ; in particular, a Moore family is never empty. The property is also called the model intersection property, since whenever we take a *meet* of a number of models we still get a model.

Assume cls has the form cl_1, \dots, cl_s , and let $\Delta = \{\varrho : \prod_k \mathcal{R}_{/k} \rightarrow \mathcal{U}^k \rightarrow \mathcal{L}\}$ denote the set of interpretations ϱ of predicate symbols in \mathcal{R} . We also define the lexicographical ordering \preceq such that $\varrho_1 \preceq \varrho_2$ if and only if there is some $1 \leq j \leq s$, where s is the order of the formula, such that the following properties hold:

- (a) $\varrho_1(R) = \varrho_2(R)$ for all $R \in \mathcal{R}$ with $\text{rank}(R) < j$,
- (b) $\varrho_1(R) \sqsubseteq \varrho_2(R)$ for all $R \in \mathcal{R}$ with $\text{rank}(R) = j$,
- (c) either $j = s$ or $\varrho_1(R) \sqsubset \varrho_2(R)$ for at least one $R \in \mathcal{R}$ with $\text{rank}(R) = j$.

We say that $\varrho_1(R) \sqsubseteq \varrho_2(R)$ if and only if $\forall \mathbf{a} \in \mathcal{U}^k : \varrho_1(R)(\mathbf{a}) \sqsubseteq \varrho_2(R)(\mathbf{a})$, where $k \geq 0$ is the arity of R . Notice that in the case $s = 1$, the above ordering coincides with lattice ordering \sqsubseteq . Intuitively, the lexicographical ordering \preceq orders the relations strata by strata starting with the strata 0. It is essentially analogous to the lexicographical ordering on strings, which is based on the alphabetical order of their characters.

Lemma 1. \preceq defines a partial order.

Proof. See Appendix A.

Assume cls has the form cl_1, \dots, cl_s where cl_j is the clause corresponding to stratum j , and let \mathcal{R}_j denote the set of all relation symbols R defined in cl_1, \dots, cl_j taking $\mathcal{R}_0 = \emptyset$. Let $M \subseteq \Delta$ denote a set of assignments which map relation symbols to relations.

Lemma 2. $\Delta = (\Delta, \preceq)$ is a complete lattice with the greatest lower bound given by

$$\left(\bigcap_{\Delta} M\right)(R) = \lambda \mathbf{a}. \bigcap \left\{ \varrho(R)(\mathbf{a}) \mid \varrho \in M_{\text{rank}(R)} \right\}$$

where

$$M_j = \left\{ \varrho \in M \mid \forall R' \text{ rank}(R') < j : \varrho(R') = \left(\bigcap_{\Delta} M\right)(R') \right\}$$

Proof. See Appendix B

Note that $\bigcap_{\Delta} M$ is well defined by induction on j observing that $M_0 = M$ and $M_j \subseteq M_{j-1}$.

Proposition 1. Assume cls is a stratified LLFP clause sequence, ς_0 and ζ_0 are interpretations of free variables and function symbols in cls , respectively. Furthermore, ϱ_0 is an interpretation of all relations of rank 0. Then $\{\varrho \mid (\varrho, \zeta_0, \varsigma_0) \models_{\beta} cls \wedge \forall R : \text{rank}(R) = 0 \Rightarrow \varrho_0(R) \sqsubseteq \varrho(R)\}$ is a Moore family.

Proof. See Appendix C

The result ensures that the approach falls within the framework of Abstract Interpretation [7,8]; hence we can be sure that there always is a single best solution for the analysis problem under consideration, namely the one defined in Proposition 1.

5 The Algorithm

In this section we present the algorithm for solving LLFP clause sequences, which extends the differential worklist algorithm by Nielson et al. [18,17]. The algorithm computes the relations in increasing order on their rank and therefore the negations present no obstacles. It completely abandons a worklist-like data structures, which are typical for most classical iterative fixpoint algorithms [12]. Instead, we adapt the recursive topdown approach of Le Charlier and van Hentenryck [6] which is enhanced by continuation based semi-naïve iteration [3,11].

In the following we assume that prior to solving the LLFP formula, all the clauses are transformed into a form such that all *applied* occurrences of variables $Y \in \mathcal{Y}$ in preconditions, i.e. $Y(u)$, are not followed by their *defining* occurrences, i.e. $R(\mathbf{u}; Y)$ and $\neg R(\mathbf{u}; Y)$. This is necessary to correctly perform late bindings of variables $Y \in \mathcal{Y}$ in the presence of $Y(u)$ construct.

The algorithm operates with (intermediate) representations of the two interpretations ς and ϱ of the semantics; we shall call them **env** and **result**, respectively, in the following. The data structure **env** is supplied as a parameter to the functions of the algorithms, and it represents partial environment. The data structure **result** is an imperative data structure that is updated as we progress.

The partial environment **env** is implemented as a map from variables to their optional values. In the case the variable is undefined it is mapped into *None*. Otherwise, depending on its type it is mapped to *Some(a)* or *Some(l)*, which means that the variable is bound to $a \in \mathcal{U}$, or $l \in \mathcal{L}_{\neq \perp}$, respectively. The main operation on **env** is the function UNIFY, defined as follows

$$\text{UNIFY}(\beta, \mathbf{env}, (\mathbf{u}; V), (\mathbf{a}; l)) = \begin{cases} \emptyset & \text{if } \text{UNIFY}_U(\mathbf{env}, \mathbf{u}, \mathbf{a}) = \text{fail} \\ \text{UNIFY}_L(\beta, \mathbf{env}', V, l) & \text{if } \text{UNIFY}_U(\mathbf{env}, \mathbf{u}, \mathbf{a}) = \mathbf{env}' \end{cases}$$

It uses two auxiliary functions that perform unifications on each component of the relation. For the first component, which ranges over the universe \mathcal{U} , the function is given by

$$\text{UNIFY}_U(\mathbf{env}, u, a) = \begin{cases} \mathbf{env} & \text{if } (u \in \mathcal{X} \wedge \mathbf{env}[u] = \text{Some}(a)) \vee u = a \\ \mathbf{env}[u \mapsto \text{Some}(a)] & \text{if } u \in \mathcal{X} \wedge \mathbf{env}[u] = \text{None} \\ \text{fail} & \text{otherwise} \end{cases}$$

It performs a unification of an argument u with an element $a \in \mathcal{U}$ in the environment **env**. In the case when the unification succeeds the modified environment

is returned, otherwise the function fails. The function is extended to k -tuples in a straightforward way. The definition of the unification function for the lattice component is given by

$$\text{UNIFY}_L(\beta, \mathbf{env}, V, l) = \begin{cases} \{\mathbf{env}[V \mapsto \text{Some}(l \sqcap l_V)]\} & \text{if } V \in \mathcal{Y} \wedge \mathbf{env}[V] = \text{Some}(l_V) \wedge l \sqcap l_V \neq \perp \\ \{\mathbf{env}[V \mapsto \text{Some}(l)]\} & \text{if } V \in \mathcal{Y} \wedge \mathbf{env}[V] = \text{None} \wedge l \neq \perp \\ \{\mathbf{env}\} & \text{if } V = [u] \wedge \\ & ((u \in \mathcal{X} \wedge \mathbf{env}[u] = \text{Some}(a)) \vee u = a) \wedge \beta(a) \sqsubseteq l \\ \{\mathbf{env}[u \mapsto \text{Some}(a)] \mid \beta(a) \sqsubseteq l\} & \text{if } V = [u] \wedge u \in \mathcal{X} \wedge \mathbf{env}[u] = \text{None} \\ \emptyset & \text{otherwise} \end{cases}$$

The function is parametrized with $\beta : \mathcal{U} \rightarrow \mathcal{L}$, defined in Section 3. It performs a unification of an lattice term V with an element $l \in \mathcal{L}$ in the environment \mathbf{env} . In the case when the unification succeeds the set of unified environments is returned, otherwise the function returns empty set.

The other important operation on the partial environment is given by the function UNIFIABLE . The function when applied to \mathbf{env} and a tuple $(\mathbf{u}; V)$, returns a set of tuples for which UNIFY would succeed. The function is defined by means of two auxiliary functions, formally we have

$$\text{UNIFIABLE}(\mathbf{env}, (\mathbf{u}; V)) = (\text{UNIFIABLE}_U(\mathbf{env}, \mathbf{u}); \text{UNIFIABLE}_L(\mathbf{env}, V))$$

where

$$\text{UNIFIABLE}_U(\mathbf{env}, u) = \begin{cases} \{a\} & \text{if } (u \in \mathcal{X} \wedge \mathbf{env}[u] = \text{Some}(a)) \vee u = a \\ \mathcal{U} & \text{if } u \in \mathcal{X} \wedge \mathbf{env}[u] = \text{None} \end{cases}$$

and

$$\text{UNIFIABLE}_L(\mathbf{env}, V) = \begin{cases} l & \text{if } V \in \mathcal{Y} \wedge \mathbf{env}[V] = \text{Some}(l) \\ \top & \text{if } V \in \mathcal{Y} \wedge \mathbf{env}[V] = \text{None} \\ \beta(a) & \text{if } V = [u] \wedge (u = a \vee (u \in \mathcal{X} \wedge \mathbf{env}[u] = \text{Some}(a))) \\ \bigsqcup \{\beta(a) \mid a \in \mathcal{U}\} & \text{if } V = [u] \wedge u \in \mathcal{X} \wedge \mathbf{env}[u] = \text{None} \\ \llbracket f \rrbracket(l) & \text{if } V = f(\mathbf{V}) \wedge l = \text{UNIFIABLE}_L(\mathbf{env}, \mathbf{V}) \end{cases}$$

Both auxiliary functions are extended to k -tuples in a straightforward way.

The global data structure **result**, which is updated incrementally during computations, is represented as a mapping from predicate names to the prefix trees that for each predicate R record the tuples currently known to belong to R . There are three main operations on the data structure **result**: the operation **result.HAS** checks whether a given tuple is associated with a given predicate, the operation **result.SUB** returns a list of the tuples associated with a given

predicate and the operation `result.ADD` adds a tuple to the interpretation of a given predicate.

Since ϱ is updated as the algorithm progresses, it may happen that a query $R(\mathbf{v}; V)$ inside a precondition fails to be satisfied at the given point in time, but may hold in the future when a new tuple $(\mathbf{a}; l)$ is added to the interpretation of R . If we are not careful we may lose the consequences that adding $(\mathbf{a}; l)$ to R will have on the contents of other predicates. This gives rise to the data structure `infl` that records computations that have to be resumed for the new tuples; these future computations are called consumers. The `infl` data structure is also represented as a mapping from the predicate names to prefix trees that for each predicate R record consumers that have to be resumed when the interpretation of R is updated. There are two main operations on the data structure `infl`: the operation `infl.REGISTER` that adds a new consumer for a given predicate and `infl.CONSUMERS` that returns all the consumers currently associated with a given predicate.

In the algorithm, we have one function for each of the three syntactic categories. The function `SOLVE` takes a clause sequence as input and calls the function `EXECUTE` on each of the individual clauses

$$\text{SOLVE}(cl_1, \dots, cl_s) = \text{EXECUTE}(cl_1)[\] ; \dots ; \text{EXECUTE}(cl_s)[\]$$

where we write $[\]$ for the empty environment reflecting that we have no free variables in the clause sequences.

Let us now turn to the description of the function `EXECUTE`. The function takes a clause cl as a parameter and a representation `env` of the interpretation of the variables. We have one case for each of the forms of cl ; the pseudo code is given in Figure 1. Let us explain the case of an assertion first. The algorithm

```

EXECUTE( $R(\mathbf{v}; V)$ )env =
  let ITERFUN ( $\mathbf{a}; l$ ) =
    match result.HAS( $R, (\mathbf{a}; l)$ ) with
    | true  $\rightarrow$  ()
    | false  $\rightarrow$ 
      result.ADD( $R, (\mathbf{a}; l)$ )
      ITER (fun  $f \rightarrow f(\mathbf{a}; l)$ ) (infl.CONSUMERS  $R$ )
  in ITER ITERFUN (UNIFIABLE(env,  $\mathbf{v}; V$ ))
EXECUTE(1)env = ()
EXECUTE( $cl_1 \wedge cl_2$ )env = EXECUTE( $cl_1$ )env; EXECUTE( $cl_2$ )env
EXECUTE( $pre \Rightarrow cl$ )env = CHECK( $pre$ , EXECUTE( $cl$ ))env
EXECUTE( $\forall x : cl$ )env = EXECUTE( $cl$ )(env[x  $\mapsto$  None])

```

Fig. 1. The `EXECUTE` function.

uses the auxiliary function `ITER`, which applies the function `ITERFUN` to each element of the list of tuples that can be unified with the argument $(\mathbf{v}; V)$. Given

a tuple $(a; l)$, the function `ITERFUN` adds the tuple to the interpretation of R stored in `result` if it is not already present. If the `ADD` operation succeeds, we first create a list of all the consumers currently registered for predicate R by calling the function `infl.CONSUMERS`. Thereafter, we resume the computations by iterating over the list of consumers and calling corresponding continuations. The cases of always true clause, `1`, is straightforward; the function simply returns the unit, without performing any other actions. In the case of the conjunction of clauses the algorithm calls the `EXECUTE` function for both conjuncts and the current environment `env`. In the case of implication we make use of the function `CHECK` that in addition to the precondition and the environment also takes the continuation `EXECUTE(cl)` as an argument. In the case of universal quantification, we simply extend the environment to record that the value of the new variable is unknown and then we recurse. The case of universal quantification over a variable $Y \in \mathcal{Y}$ is exactly the same and hence omitted.

Now, let us present the function `CHECK`. It takes a precondition, a continuation and an environment as parameters. The pseudo code is given in Figure 2. In the case of positive queries we first ensure that the consumer is registered

```

CHECK( $R(v; V), next$ )env =
  let CONSUMER ( $a; l$ ) =
    match UNIFY(env, ( $v; V$ ), ( $a; l$ )) with
    | fail  $\rightarrow$  ()
    | envs  $\rightarrow$  ITER next envs
  in infl.REGISTER( $R$ , CONSUMER); ITER CONSUMER (result.SUB  $R$ )
CHECK( $\neg R(v; V), next$ )env =
  let ITERFUN ( $a; l$ ) =
    match result.HAS( $R$ , ( $a; l$ )) with
    | true  $\rightarrow$  ()
    | false  $\rightarrow$  ITER next (UNIFY(env, ( $v; V$ ), ( $a; l$ )))
  in ITER ITERFUN (UNIFIABLE(env, ( $v; V$ )))
CHECK( $Y(x), next$ )env =
  let env' = if env( $Y$ ) = Some( $l$ ) then env else env[ $Y \mapsto \top$ ]
  in let F a = if Some( $\beta(a)$ )  $\sqsubseteq$  env'( $Y$ ) then next env'[ $x \mapsto a$ ] else ()
  in match env'(x) with
  | Some( $a$ )  $\rightarrow$  F a
  | None  $\rightarrow$  ITER F U
CHECK( $pre_1 \wedge pre_2, next$ )env = CHECK( $pre_1$ , CHECK( $pre_2, next$ ))env
CHECK( $pre_1 \vee pre_2, next$ )env = CHECK( $pre_1, next$ )env; CHECK( $pre_2, next$ )env
CHECK( $\exists x : pre, next$ )env = CHECK( $pre, next \circ (\text{REMOVE } x)$ )(env[ $x \mapsto \text{None}$ ])

```

Fig. 2. The `CHECK` function.

in `infl`, by calling function `REGISTER`, so that future tuples associated with R will be processed. Thereafter, the function inspects the data structure `result` to obtain the list of tuples associated with the predicate R . Then, the auxiliary function `CONSUMER` unifies $(v; V)$ with each tuple; and if the operation succeeds,

the continuation *next* is invoked on each of the updated new environments in the returned set **envs**. In the case of negated query, the algorithm first computes the tuples unifiable with $(v; V)$ in the environment **env**. Then, for each tuple it checks whether the tuple is already in R and if not, the tuple is unified with $(v; V)$ to produce set of new environments. Thereafter, the continuation *next* is evaluated in each of the environments contained in the returned set. Notice that in the case of negative queries we do not register a consumer for the relation R . This is because the stratification condition introduced in Definition 3 ensures that the relation is fully evaluated before it is queried negatively. Thus, there is no need to register future computations since the interpretation of R will not change. Now, let us consider function CHECK in the case of $Y(x)$, where $x \in \mathcal{X}$. The function begins with creating an environment **env'** that is exactly as **env** except that the binding for the variable Y is set to \top in the case Y is undefined in **env**. Then, we define an auxiliary function that checks whether **env'**(Y) over-approximates the abstraction of an argument a , denoted by $\beta(a)$, and if so the continuation is called in the environment **env'**[$x \mapsto a$]. Finally, the function checks the binding for the variable x in the environment **env'** and if it is bound to *Some*(a) the function F applied to a is called. Otherwise, the function F is called for each element of the universe, using the ITER function. The case of $Y(a)$, where $a \in \mathcal{U}$ is essentially the same as the case explained above, except that we do not have to handle the case when $x \in \mathcal{X}$ is undefined in **env**. For conjunction of preconditions we exploit a continuation passing programming style. More precisely, we call the CHECK function for the precondition pre_1 , and as a continuation we pass a call to the CHECK function partially applied to the precondition pre_2 and the continuation *next*. In the case of disjunction of preconditions the function simply checks preconditions pre_1 and pre_2 respectively in the current environment **env**. In order to be efficient we use memoization; this means that if both checks yield the same bindings of variables, the second check does not need to consider the continuation, as it has already been done. The algorithm for existential quantification checks the precondition pre in the environment extended with the quantified variable. The continuation that is passed is a composition of functions *next* and REMOVE x , where the function REMOVE removes variable passed as a first argument from the environment passed as a second argument. In order to be efficient we again use a memoization to avoid redundant computations. The case of existential quantification over a variable $Y \in \mathcal{Y}$ is exactly the same and hence omitted.

6 Conclusions and Future Work

In the paper we introduced the LLFP logic, which is an expressive formalism for specifying static analysis problems. It lifts the limitation of logics such as Datalog and ALFP by allowing interpretation over complete lattices satisfying Ascending Chain Condition. Thanks to the declarative style, the analysis specifications are easy to analyse for their correctness.

We established a Moore Family result that guarantees that there always is a unique best solution for the LLFP formulae. More generally this ensures that the approach taken falls within the general Abstract Interpretation framework. We also developed a state-of-the-art solving algorithm for LLFP, which is a continuation passing style algorithm, which represents relations as prefix trees. We showed that the logic and the associated solver can be used for rapid prototyping of sophisticated static analyses by presenting the formulation of interval analysis.

As a future work we plan to implement a front-end to automatically extract analysis relations from program source code, and perform experiments on real-world programs in order to evaluate the performance of the LLFP solver. Furthermore, we would like to lift the Ascending Chain Condition and use e.g. *widening operator* [9,10] in order to ensure termination of the least fixed point computation.

References

1. K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In Foundations of Deductive Databases and Logic Programming., pages 89–148. Morgan Kaufmann, 1988.
2. C. Baier and J.-P. Katoen. Principles of Model Checking (Representation and Mind Series). The MIT Press, 2008.
3. I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. Journal of Logic Programming, 4(3):259–262, 1987.
4. M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In S. Arora and G. T. Leavens, editors, OOPSLA, pages 243–262. ACM, 2009.
5. A. K. Chandra and D. Harel. Computable queries for relational data bases (preliminary report). In STOC, pages 309–318, 1979.
6. B. L. Charlier and P. V. Hentenryck. A universal top-down fixpoint algorithm. Technical report, CS-92-25, Brown University, 1992.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL, pages 238–252, 1977.
8. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In POPL, pages 269–282, 1979.
9. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. J. Log. Program., 13(2&3):103–179, 1992.
10. P. Cousot and R. Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, PLILP, volume 631 of Lecture Notes in Computer Science, pages 269–295. Springer, 1992.
11. C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. Nordic Journal of Computing, 5(4):304–329, 1998.
12. C. Fecht and H. Seidl. A faster solver for general systems of equations. Sci. Comput. Program., 35(2):137–161, 1999.

13. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. Acta Inf., 7:305–317, 1977.
14. G. A. Kildall. A unified approach to global program optimization. In POPL, pages 194–206, 1973.
15. M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In PODS, pages 1–12, 2005.
16. F. Nielson, H. R. Nielson, and C. Hankin. Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
17. F. Nielson, H. R. Nielson, H. Sun, M. Buchholtz, R. R. Hansen, H. Pilegaard, and H. Seidl. The succinct solver suite. In TACAS, pages 251–265, 2004.
18. F. Nielson, H. Seidl, and H. R. Nielson. A Succinct Solver for ALFP. Nord. J. Comput., 9(4):335–372, 2002.
19. T. W. Reps. Demand interprocedural program analysis using logic databases. In Workshop on Programming with Logic Databases (Book), ILPS, pages 163–196, 1993.
20. J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog with binary decision diagrams for program analysis. In APLAS, pages 97–118, 2005.
21. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In PLDI, pages 131–144, 2004.

These appendices are not intended for publication and references to them will be removed in the final version.

A Proof of Lemma 1

Proof. **Reflexivity** $\forall \varrho \in \Delta : \varrho \preceq \varrho$.

To show that $\varrho \preceq \varrho$ let us take $j = s$. If $\text{rank}(R) < j$ then $\varrho(R) = \varrho(R)$ as required. Otherwise if $\text{rank}(R) = j$ then from $\varrho(R) = \varrho(R)$ we get $\varrho(R) \sqsubseteq \varrho(R)$. Thus we get the required $\varrho \preceq \varrho$.

Transitivity $\forall \varrho_1, \varrho_2, \varrho_3 \in \Delta : \varrho_1 \preceq \varrho_2 \wedge \varrho_2 \preceq \varrho_3 \Rightarrow \varrho_1 \preceq \varrho_3$.

Let us assume that $\varrho_1 \preceq \varrho_2 \wedge \varrho_2 \preceq \varrho_3$. From $\varrho_i \preceq \varrho_{i+1}$ we have j_i such that conditions (a)–(c) are fulfilled for $i = 1, 2$. Let us take j to be the minimum of j_1 and j_2 . Now we need to verify that conditions (a)–(c) hold for j . If $\text{rank}(R) < j$ we have $\varrho_1(R) = \varrho_2(R)$ and $\varrho_2(R) = \varrho_3(R)$. It follows that $\varrho_1(R) = \varrho_3(R)$, hence (a) holds. Now let us assume that $\text{rank}(R) = j$. We have $\varrho_1(R) \sqsubseteq \varrho_2(R)$ and $\varrho_2(R) \sqsubseteq \varrho_3(R)$ and from transitivity of \sqsubseteq we get $\varrho_1(R) \sqsubseteq \varrho_3(R)$, which gives (b). Let us now assume that $j \neq s$, hence $\varrho_i(R) \sqsubset \varrho_{i+1}(R)$ for some $R \in \mathcal{R}$ and $i = 1, 2$. Without loss of generality let us assume that $\varrho_1(R) \sqsubset \varrho_2(R)$. We have $\varrho_1(R) \sqsubset \varrho_2(R)$ and $\varrho_2(R) \sqsubseteq \varrho_3(R)$, hence $\varrho_1(R) \sqsubset \varrho_3(R)$, and (c) holds.

Anti-symmetry $\forall \varrho_1, \varrho_2 \in \Delta : \varrho_1 \preceq \varrho_2 \wedge \varrho_2 \preceq \varrho_1 \Rightarrow \varrho_1 = \varrho_2$.

Let us assume $\varrho_1 \preceq \varrho_2$ and $\varrho_2 \preceq \varrho_1$. Let j be minimal such that $\text{rank}(R) = j$ and $\varrho_1(R) \neq \varrho_2(R)$ for some $R \in \mathcal{R}$. Then, since $\text{rank}(R) = j$, we have $\varrho_1(R) \sqsubseteq \varrho_2(R)$ and $\varrho_2(R) \sqsubseteq \varrho_1(R)$. Hence $\varrho_1(R) = \varrho_2(R)$ which is a contradiction. Thus it must be the case that $\varrho_1(R) = \varrho_2(R)$ for all $R \in \mathcal{R}$. \square

B Proof of Lemma 2

Proof. First we prove that $\sqcap_{\Delta} M$ is a lower bound of M ; that is $\sqcap_{\Delta} M \preceq \varrho$ for all $\varrho \in M$. Let j be maximum such that $\varrho \in M_j$; since $M = M_0$ and $M_j \supseteq M_{j+1}$ clearly such j exists. From definition of M_j it follows that $(\sqcap_{\Delta} M)(R) = \varrho(R)$ for all R with $\text{rank}(R) < j$; hence (a) holds. If $\text{rank}(R) = j$ we have $(\sqcap_{\Delta} M)(R) = \lambda \mathbf{a} . \sqcap \{ \varrho'(R)(\mathbf{a}) \mid \varrho' \in M_j \} \sqsubseteq \varrho(R)$ showing that (b) holds. Finally let us assume that $j \neq s$; we need to show that there is some R with $\text{rank}(R) = j$ such that $(\sqcap_{\Delta} M)(R) \sqsubset \varrho(R)$. Since we know that j is maximum such that $\varrho \in M_j$, it follows that $\varrho \notin M_{j+1}$, hence there is a relation R with $\text{rank}(R) = j$ such that $(\sqcap_{\Delta} M)(R) \sqsubset \varrho(R)$; thus (c) holds.

Now we need to show that $\sqcap_{\Delta} M$ is the greatest lower bound. Let us assume that $\varrho' \preceq \varrho$ for all $\varrho \in M$, and let us show that $\varrho' \preceq \sqcap_{\Delta} M$. If $\varrho' = \sqcap_{\Delta} M$ the result holds vacuously, hence let us assume $\varrho' \neq \sqcap_{\Delta} M$. Then there exists a minimal j such that $(\sqcap_{\Delta} M)(R) \neq \varrho'(R)$ for some R with $\text{rank}(R) = j$. Let us first consider R such that $\text{rank}(R) < j$. By our choice of j we have $(\sqcap_{\Delta} M)(R) = \varrho'(R)$ hence (a) holds. Next assume that $\text{rank}(R) = j$. Since we assumed that $\varrho' \preceq \varrho$ for all $\varrho \in M$ and $M_j \subseteq M$, it follows that $\varrho'(R) \sqsubseteq \varrho(R)$ for all $\varrho \in M_j$. Thus we have $\varrho'(R) \sqsubseteq \lambda \mathbf{a} . \sqcap \{ \varrho(R)(\mathbf{a}) \mid \varrho \in M_j \}$. Since $(\sqcap_{\Delta} M)(R) = \lambda \mathbf{a} . \sqcap \{ \varrho(R)(\mathbf{a}) \mid \varrho \in$

$M_j\}$, we have $\varrho'(R) \sqsubseteq (\sqcap_{\Delta} M)(R)$ which proves (b). Finally since we assumed that $\varrho'(R) \neq (\sqcap_{\Delta} M)(R)$ for some R with $\text{rank}(R) = j$, it follows that (c) holds. Thus we proved that $\varrho' \preceq \sqcap_{\Delta} M$. \square

C Proof of Proposition 1

In order to prove Proposition 1 we first state and prove two auxiliary lemmas.

Lemma 3. *If $\varrho = \sqcap_{\Delta} M$, pre occurs in cl_j and $(\varrho, \varsigma) \models_{\beta} pre$ then also $(\varrho', \varsigma) \models_{\beta} pre$ for all $\varrho' \in M_j$.*

Proof. We proceed by induction on j and in each case perform a structural induction on the form of the precondition pre occurring in cl_j .

Case: $pre = R(\mathbf{u}; V)$

Let us take $\varrho = \sqcap_{\Delta} M$ and assume that

$$(\varrho, \varsigma) \models_{\beta} R(\mathbf{u}; V)$$

From Table 1 we have:

$$\varrho(R)(\varsigma(\mathbf{u})) \supseteq \varsigma(V)$$

Depending on the rank of R we have two cases. If $\text{rank}(R) = j$ then $\varrho(R) = \lambda \mathbf{a}. \sqcap \{\varrho'(R)(\mathbf{a}) \mid \varrho' \in M_j\}$ and hence we have

$$\sqcap \{\varrho'(R)(\varsigma(\mathbf{u})) \mid \varrho' \in M_j\} \supseteq \varsigma(V)$$

It follows that for all $\varrho' \in M_j$

$$\varrho'(R)(\varsigma(\mathbf{u})) \supseteq \varsigma(V)$$

Now if $\text{rank}(R) < j$ then $\varrho(R) = \varrho'(R)$ for all $\varrho' \in M_j$ hence we have that for all $\varrho' \in M_j$

$$\varrho'(R)(\varsigma(\mathbf{u})) \supseteq \varsigma(V)$$

which according to Table 1 is equivalent to

$$\forall \varrho' \in M_j : (\varrho', \varsigma) \models_{\beta} R(\mathbf{u}; V)$$

which was required and finishes the case.

Case: $pre = Y(u)$

Let us take $\varrho = \sqcap_{\Delta} M$ and assume that

$$(\varrho, \varsigma) \models_{\beta} Y(u)$$

According to the semantics of LLFP in Table 1 we have

$$\beta(\varsigma(u)) \sqsubseteq \varsigma(Y)$$

It follows that

$$\forall \varrho' \in M_j : \beta(\varsigma(u)) \sqsubseteq \varsigma(Y)$$

which according to the semantics of LLFP in Table 1 is equivalent to

$$\forall \varrho' \in M_j : (\varrho', \varsigma) \models_{\beta} Y(u)$$

which was required and finishes the case.

Case: $pre = \neg R(\mathbf{u}; V)$

Let us take $\varrho = \prod_{\Delta} M$ and assume that

$$(\varrho, \varsigma) \models_{\beta} \neg R(\mathbf{u}; V)$$

From Table 1 we have:

$$\mathbb{C}(\varrho(R)(\varsigma(\mathbf{u}))) \supseteq \varsigma(V)$$

Since $\text{rank}(R) < j$ then we know that $\varrho(R) = \varrho'(R)$ for all $\varrho' \in M_j$ hence we have that

$$\forall \varrho' \in M_j : \mathbb{C}(\varrho(R)(\varsigma(\mathbf{u}))) \supseteq \varsigma(V)$$

Which according to Table 1 is equivalent to

$$\forall \varrho' \in M_j : (\varrho', \varsigma) \models_{\beta} \neg R(\mathbf{u}; V)$$

which was required and finishes the case.

Case: $pre = pre_1 \wedge pre_2$

Let us take $\varrho = \prod_{\Delta} M$ and assume that

$$(\varrho, \varsigma) \models_{\beta} pre_1 \wedge pre_2$$

According to Table 1 we have

$$(\varrho, \varsigma) \models_{\beta} pre_1$$

and

$$(\varrho, \varsigma) \models_{\beta} pre_2$$

From the induction hypothesis we get that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_{\beta} pre_1$$

and

$$(\varrho', \varsigma) \models_{\beta} pre_2$$

It follows that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_{\beta} pre_1 \wedge pre_2$$

which was required and finishes the case.

Case: $pre = pre_1 \vee pre_2$

Let us take $\varrho = \prod_{\Delta} M$ and assume that

$$(\varrho, \varsigma) \models_{\beta} pre_1 \vee pre_2$$

According to Table 1 we have

$$(\varrho, \varsigma) \models_{\beta} pre_1$$

or

$$(\varrho, \varsigma) \models_{\beta} pre_2$$

From the induction hypothesis we get that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_{\beta} pre_1$$

or

$$(\varrho', \varsigma) \models_{\beta} pre_2$$

It follows that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_{\beta} pre_1 \vee pre_2$$

which was required and finishes the case.

Case: $pre = \exists x : pre'$

Let us take $\varrho = \prod_{\Delta} M$ and assume that

$$(\varrho, \varsigma) \models_{\beta} \exists x : pre'$$

According to Table 1 we have

$$\exists a \in \mathcal{U} : (\varrho, \varsigma[x \mapsto a]) \models_{\beta} pre'$$

From the induction hypothesis we get that for all $\varrho' \in M_j$

$$\exists a \in \mathcal{U} : (\varrho', \varsigma[x \mapsto a]) \models_{\beta} pre'$$

It follows from Table 1 that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_{\beta} \exists x : pre'$$

which was required and finishes the case.

Case: $pre = \exists Y : pre'$

Let us take $\varrho = \prod_{\Delta} M$ and assume that

$$(\varrho, \varsigma) \models_{\beta} \exists Y : pre'$$

According to Table 1 we have

$$\exists l \in \mathcal{L}_{\neq \perp} : (\varrho, \varsigma[Y \mapsto l]) \models_{\beta} pre'$$

From the induction hypothesis we get that for all $\varrho' \in M_j$

$$\exists l \in \mathcal{L}_{\neq \perp} : (\varrho', \varsigma[Y \mapsto l]) \models_{\beta} pre'$$

It follows from Table 1 that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_{\beta} \exists Y : pre'$$

which was required and finishes the case. □

Lemma 4. *If $\varrho = \prod_{\Delta} M$ and $(\varrho', \zeta, \varsigma) \models_{\beta} cl_j$ for all $\varrho' \in M$ then $(\varrho, \zeta, \varsigma) \models_{\beta} cl_j$.*

Proof. We proceed by induction on j and in each case perform a structural induction on the form of the clause occurring in cl_j .

Case: $cl_j = R(\mathbf{u}; V)$

Assume that for all $\varrho' \in M$

$$(\varrho', \zeta, \varsigma) \models_{\beta} R(\mathbf{u}; V)$$

From the semantics of LLFP we have that for all $\varrho' \in M$

$$\varrho'(R)(\llbracket \mathbf{u} \rrbracket(\zeta, \varsigma)) \supseteq \llbracket V \rrbracket(\zeta, \varsigma)$$

It follows that:

$$\prod \{ \varrho'(R)(\llbracket \mathbf{u} \rrbracket(\zeta, \varsigma)) \mid \varrho' \in M \} \supseteq \llbracket V \rrbracket(\zeta, \varsigma)$$

Since $M_j \subseteq M$, we have:

$$\prod \{ \varrho'(R)(\llbracket \mathbf{u} \rrbracket(\zeta, \varsigma)) \mid \varrho' \in M_j \} \supseteq \llbracket V \rrbracket(\zeta, \varsigma)$$

We know that $\text{rank}(R) = j$; hence $\varrho(R) = \lambda \mathbf{a}. \prod \{ \varrho'(R)(\mathbf{a}) \mid \varrho' \in M_j \}$; thus

$$\varrho(R)(\llbracket \mathbf{u} \rrbracket(\zeta, \varsigma)) = \prod \{ \varrho'(R)(\llbracket \mathbf{u} \rrbracket(\zeta, \varsigma)) \mid \varrho' \in M_j \} \supseteq \llbracket V \rrbracket(\zeta, \varsigma)$$

Which according to Table 1 is equivalent to

$$(\varrho, \zeta, \varsigma) \models_{\beta} R(\mathbf{u}; V)$$

Case: $cl_j = cl_1 \wedge cl_2$

Assume that for all $\varrho' \in M$:

$$(\varrho', \zeta, \varsigma) \models_{\beta} cl_1 \wedge cl_2$$

From Table 1 it is equivalent to

$$(\varrho', \zeta, \varsigma) \models_{\beta} cl_1 \text{ and } (\varrho', \zeta, \varsigma) \models_{\beta} cl_2$$

The induction hypothesis gives that

$$(\varrho, \zeta, \varsigma) \models_{\beta} cl_1 \text{ and } (\varrho, \zeta, \varsigma) \models_{\beta} cl_2$$

Which according to Table 1 is equivalent to

$$(\varrho, \zeta, \varsigma) \models_{\beta} cl_1 \wedge cl_2$$

and finishes the case.

Case: $cl_j = pre \Rightarrow cl$

Assume that for all $\varrho' \in M$:

$$(\varrho', \zeta, \varsigma) \models_{\beta} pre \Rightarrow cl \tag{1}$$

We have two cases. In the first one $(\varrho, \varsigma) \models_{\beta} pre$ is *false*, hence $(\varrho, \varsigma, \zeta) \models_{\beta} pre \Rightarrow cl$ holds trivially. In the second case let us assume:

$$(\varrho, \varsigma) \models_{\beta} pre \quad (2)$$

Lemma 3 gives that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_{\beta} pre$$

From (1) we have that for all $\varrho' \in M_j$

$$(\varrho', \zeta, \varsigma) \models_{\beta} cl$$

and the induction hypothesis gives:

$$(\varrho, \zeta, \varsigma) \models_{\beta} cl$$

Hence from (2) we get:

$$(\varrho, \zeta, \varsigma) \models_{\beta} pre \Rightarrow cl$$

which was required and finishes the case.

Case: $cl_j = \forall x : cl$

Assume that for all $\varrho' \in M$

$$(\varrho', \zeta, \varsigma) \models_{\beta} \forall x : cl$$

From Table 1 we have that for all $\varrho' \in M$ and for all $a \in \mathcal{U}$

$$(\varrho', \zeta, \varsigma[x \mapsto a]) \models_{\beta} cl$$

Thus from the induction hypothesis we get that for all $a \in \mathcal{U}$

$$(\varrho, \zeta, \varsigma[x \mapsto a]) \models_{\beta} cl$$

According to Table 1 it is equivalent to

$$(\varrho, \zeta, \varsigma) \models_{\beta} \forall x : cl$$

which was required and finishes the case.

Case: $cl = \forall Y : cl$

Assume that for all $\varrho' \in M$

$$(\varrho', \zeta, \varsigma) \models_{\beta} \forall Y : cl$$

From Table 1 we have that $\varrho' \in M$

$$\forall l \in \mathcal{L}_{\neq \perp} : (\varrho', \zeta, \varsigma[Y \mapsto l]) \models_{\beta} cl$$

Thus from the induction hypothesis we get that

$$\forall l \in \mathcal{L}_{\neq \perp} : (\varrho, \zeta, \varsigma[Y \mapsto l]) \models_{\beta} cl$$

According to Table 1 it is equivalent to

$$(\varrho, \zeta, \varsigma) \models_{\beta} \forall Y : cl$$

which was required and finishes the case.

Proposition 1. Assume cls is a stratified LLFP clause sequence, ς_0 and ζ_0 are interpretations of free variables and function symbols in cls , respectively. Furthermore, ϱ_0 is an interpretation of all relations of rank 0. Then $\{\varrho \mid (\varrho, \zeta_0, \varsigma_0) \models_{\beta} cls \wedge \forall R : \text{rank}(R) = 0 \Rightarrow \varrho_0(R) \subseteq \varrho(R)\}$ is a Moore family.

Proof. The result follows from Lemma 4. □